

# Optimised determinisation and completion of finite tree automata

John P. Gallagher<sup>a,b,\*</sup>, Mai Ajspur<sup>c</sup>, Bishoksan Kaffle<sup>d</sup>

<sup>a</sup>*Roskilde University, Denmark*

<sup>b</sup>*IMDEA Software Institute, Madrid, Spain*

<sup>c</sup>*IT University of Copenhagen, Denmark*

<sup>d</sup>*The University of Melbourne, Australia*

---

## Abstract

Determinisation and completion of finite tree automata are important operations with applications in program analysis and verification. However, the complexity of the classical procedures for determinisation and completion is high. They are not practical procedures for manipulating tree automata beyond very small ones. In this paper we develop an algorithm for determinisation and completion of finite tree automata, whose worst-case complexity remains unchanged, but which performs far better than existing algorithms in practice. The critical aspect of the algorithm is that the transitions of the determinised (and possibly completed) automaton are generated in a potentially very compact form called product form, which can reduce the size of the representation dramatically. Furthermore, the representation can often be used directly when manipulating the determinised automaton. The paper contains an experimental evaluation of the algorithm on a large set of tree automata examples.

---

## 1. Introduction

A recognisable tree language is a possibly infinite set of trees that are accepted by a finite tree automaton (FTA). FTAs and the corresponding recognisable languages have desirable properties such as closure under Boolean set operations, and decidability of membership and emptiness.

In the paper we will give a brief overview of the relevant features of FTAs, but the main goal of the paper is to focus on two operations on FTAs, namely *determinisation* and *completion*. These operations play a key role in the theory of FTAs, for example in showing that recognisable tree languages are closed under Boolean operations. Potentially, they also play a practical role in systems that manipulate sets of terms, but their complexity has so far discouraged their widespread application.

---

\*Corresponding author

Email address: [jpg@ruc.dk](mailto:jpg@ruc.dk) (John P. Gallagher)

In the paper we develop an optimised algorithm that performs determinisation and optionally completion, and analyse its properties. The most critical aspect of the optimisation is a compact representation of the set of transitions of the determinised automaton, called *product transitions*. Experiments show that the algorithm performs well, though the worst case remains unchanged. We also discuss applications of finite tree automata that exploit the determinisation algorithm.

In Section 2 the essentials, for our purposes, of finite tree automata are introduced, including the notion of product transitions. The operations of determinisation and completion are defined. Section 3 presents the optimised algorithm for determinising an FTA. It is developed in a series of stages starting from the textbook algorithm for determinisation. In Section 3 it is shown how the algorithm can be optimised and output transitions in product form. The performance of the algorithm is analysed in Section 4. In Section 5 we discuss the combination of determinisation and completion of an FTA and show that the performance of the algorithm generating product form is as effective when generating a complete determinised automaton. Section 6 reports on the performance of the algorithm on a large number of example tree automata. Section 7 discusses potential applications of the algorithm to problems in program analysis and verification, and also to tree automata problems including checking inclusion and universality., Section 8 contains a discussion of related work and finally in Section 9 we summarise and discuss further work and applications.

## 2. Preliminaries

A *finite tree automaton* (FTA) is a quadruple  $\langle Q, Q_f, \Sigma, \Delta \rangle$ , where

1.  $Q$  is a finite set called *states*,
2.  $Q_f \subseteq Q$  is called the set of accepting (or final) states,
3.  $\Sigma$  is a set of function symbols (called the *signature*) and
4.  $\Delta$  is a set of *transitions*.

Each function symbol  $f \in \Sigma$  has an arity  $n \geq 0$ , written  $\text{ar}(f) = n$ . Function symbols with arity 0 are called *constants*.  $Q$  and  $\Sigma$  are disjoint.  $\text{Term}(\Sigma)$  is the set of *ground terms* (also called *trees*) constructed from  $\Sigma$  where  $t \in \text{Term}(\Sigma)$  iff  $t \in \Sigma$  is a constant or  $t = f(t_1, \dots, t_n)$  where  $\text{ar}(f) = n$  and  $t_1, \dots, t_n \in \text{Term}(\Sigma)$ . Similarly  $\text{Term}(\Sigma \cup Q)$  is the set of terms/trees constructed from  $\Sigma$  and  $Q$ , treating the elements of  $Q$  as constants. Each transition in  $\Delta$  is of the form  $f(q_1, \dots, q_n) \rightarrow q$ , where  $\text{ar}(f) = n$  and  $q, q_1, \dots, q_n \in Q$ .

To define acceptance of a term by the FTA  $\langle Q, Q_f, \Sigma, \Delta \rangle$  we first define a *context* for the FTA. A context is a term from  $\text{Term}(\Sigma \cup Q \cup \{\bullet\})$  containing exactly one occurrence of  $\bullet$  (which is a constant not in  $\Sigma$  or  $Q$ ). Let  $c$  be a context and  $t \in \text{Term}(\Sigma \cup Q)$ ;  $c[t]$  denotes the term resulting from the replacement of  $\bullet$  in  $c$  by  $t$ . A term  $t \in \text{Term}(\Sigma \cup Q)$  can be written as  $c[t']$  if  $t$  has a subterm  $t'$ , where  $c$  is the context resulting from replacing that subtree by  $\bullet$ .

The binary relation  $\Rightarrow$  represents one step of a (bottom-up) run for the FTA. It is defined as follows;  $c[l] \Rightarrow c[r]$  iff  $c$  is a context and  $l \rightarrow r \in \Delta$ . The reflexive, transitive closure of  $\Rightarrow$  is denoted  $\Rightarrow^*$ .

A run for  $t \in \text{Term}(\Sigma)$  exists if  $t \Rightarrow^* q$  where  $q \in Q$ . The run is *successful* if  $q \in Q_f$  and in this case  $t$  is *accepted* by the FTA. There may be more than one state  $q$  such that  $t \Rightarrow^* q$  and hence FTAs are sometimes called NFTAs, where N stands for nondeterministic. A tree automaton  $R$  defines a set of terms, that is, a tree language, denoted  $L(R)$ , as the set of all terms that it accepts. We also write  $L(q)$  to be the set of terms  $t$  such that  $t \Rightarrow^* q$  in a given FTA.

**Definition 1.** An FTA  $\langle Q, Q_f, \Sigma, \Delta \rangle$  is called bottom-up deterministic if and only if  $\Delta$  contains no two transitions with the same left hand side. A bottom-up deterministic FTA is abbreviated as a DFTA.

Runs of a DFTA are deterministic in the following sense; for every context  $c$  and term of form  $c[t]$  there is at most one term  $c[t']$  such that  $c[t] \Rightarrow c[t']$ . It follows that for every  $t \in \text{Term}(\Sigma)$  there is at most one  $q \in Q$  such that  $t \Rightarrow^* q$ . As far as expressiveness is concerned we can limit our attention to DFTAs<sup>1</sup>. For every FTA  $R$  there exists a DFTA  $R'$  such that  $L(R) = L(R')$ .

**Definition 2.** An automaton  $R = \langle Q, Q_f, \Sigma, \Delta \rangle$  is complete if for all  $n$ -ary functions  $f \in \Sigma$  and states  $q_1, \dots, q_n \in Q$ , there exists a state  $q \in Q$  such that  $f(q_1, \dots, q_n) \rightarrow q \in \Delta$ .

It follows that in a complete FTA every term  $t$  has at least one run and furthermore in a complete DFTA each  $t$  has a run to exactly one state. Thus a complete DFTA defines a partition of  $\text{Term}(\Sigma)$ , namely  $\{L(q) \mid q \in Q\} \setminus \{\emptyset\}$ .

**Definition 3.** Let  $\Sigma$  be a signature and “any” a state. We define  $\Delta_{any}^\Sigma$  to be the following set of transitions.

$$\{f(\overbrace{any, \dots, any}^{n \text{ times}}) \rightarrow any \mid f^n \in \Sigma\}$$

Clearly, given an FTA  $\langle Q, Q_f, \Sigma, \Delta \rangle$  with  $any \in Q$  and  $\Delta_{any}^\Sigma \subseteq \Delta$ , there is a run  $t \Rightarrow^* any$  for any  $t \in \text{Term}(\Sigma)$ , that is,  $L(any) = \text{Term}(\Sigma)$ .

We normally drop the superscript in  $\Delta_{any}^\Sigma$  as  $\Sigma$  is usually clear from the context.

**Example 1.** Let  $\Sigma = \{\square, [\cdot|\cdot], 0\}$ ,  $Q = \{list, listlist, any\}$ ,  $Q_f = \{list, listlist\}$  and  $\Delta = \{\square \rightarrow list, [any|list] \rightarrow list, \square \rightarrow listlist, [list|listlist] \rightarrow listlist\} \cup \Delta_{any}$ .  $L(list)$  is the set of lists of any terms, while  $L(listlist)$  is the set of lists whose elements are themselves lists. Clearly  $L(listlist)$  is contained in  $L(list)$ , which is contained in  $L(any)$ .

---

<sup>1</sup>We do not deal here with top-down deterministic FTA, which are strictly less expressive than FTAs.

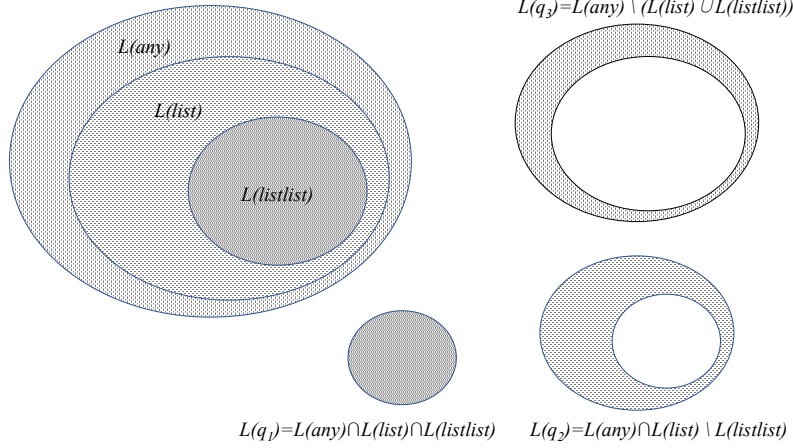


Figure 1: The disjoint languages from Example 1

The automaton is not bottom-up deterministic since  $\square$  occurs as the left hand side in more than one transition; a determinisation algorithm (see Section 3) yields the DFTA  $\langle Q', Q'_f, \Sigma, \Delta' \rangle$ , where  $Q' = \{q_1, q_2, q_3\}$ ,  $Q'_f = \{q_1, q_2\}$  and  $\Delta' = \{\square \rightarrow q_1, [q_1|q_1] \rightarrow q_1, [q_2|q_1] \rightarrow q_1, [q_1|q_2] \rightarrow q_2, [q_2|q_2] \rightarrow q_2, [q_3|q_2] \rightarrow q_2, [q_3|q_1] \rightarrow q_2, [q_2|q_3] \rightarrow q_3, [q_1|q_3] \rightarrow q_3, [q_3|q_3] \rightarrow q_3, 0 \rightarrow q_3\}$ .

The states  $q_1$ ,  $q_2$  and  $q_3$  are abbreviations for elements of the powerset of the states of the original FTA; here  $q_1 = \{\text{any}, \text{list}, \text{listlist}\}$ ,  $q_2 = \{\text{any}, \text{list}\}$  and  $q_3 = \{\text{any}\}$ . This automaton is also complete.  $\square$

In Example 1,  $L(q_1) = L(\text{any}) \cap L(\text{list}) \cap L(\text{listlist})$ ,  $L(q_2) = (L(\text{list}) \cap L(\text{any})) \setminus L(\text{listlist})$ , and  $L(q_3) = L(\text{any}) \setminus (L(\text{list}) \cup L(\text{listlist}))$ . The relationship between the languages corresponding to the FTA and DFTA states in Example 1 is shown in Figure 1.

A critical aspect of this paper is a compact representation for the set of transitions  $\Delta$ , using the notion of product transition defined as follows.

**Definition 4 (Product transition).** Let  $\langle Q, F, \Sigma, \Delta \rangle$  be an FTA. A product transition is of the form  $f(Q_1, \dots, Q_n) \rightarrow q$  where  $Q_i \subseteq Q$ ,  $1 \leq i \leq n$  and  $q \in Q$ . This product transition denotes the set of transitions  $\{f(q_1, \dots, q_n) \rightarrow q \mid q_1 \in Q_1, \dots, q_n \in Q_n\}$ . Thus  $\prod_{i=1}^n |Q_i|$  transitions are represented by a single product transition.

Instead of expanding the product transitions, we can regard a product transition as introducing  $\epsilon$ -transitions. An  $\epsilon$ -transition has the form  $q_1 \rightarrow q_2$  where  $q_1, q_2$  are states.  $\epsilon$ -transitions can be eliminated, if desired. Given a product transition  $f(Q_1, \dots, Q_n) \rightarrow q$ , introduce  $n$  new non-final states  $s_1, \dots, s_n$  corresponding to  $Q_1, \dots, Q_n$  respectively and replace the product transition by the

```

1: procedure FTA DETERMINISATION (Input:  $\langle Q, Q_f, \Sigma, \Delta \rangle$ )
2:    $\mathcal{Q}_d \leftarrow \emptyset$ 
3:    $\Delta_d \leftarrow \emptyset$ 
4:   repeat
5:      $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$ ,
6:      $\Delta_d \leftarrow \Delta_d \cup \{f(Q_1, \dots, Q_{\text{ar}(f)}) \rightarrow Q_0\}$ 
7:     where
8:        $f^n \in \Sigma, Q_1, \dots, Q_{\text{ar}(f)} \in \mathcal{Q}_d$ ,
9:        $Q_0 = \{q_0 \mid \exists q_1 \in Q_1, \dots, q_{\text{ar}(f)} \in Q_{\text{ar}(f)}, (f(q_1, \dots, q_{\text{ar}(f)}) \rightarrow q_0) \in \Delta\}$ 
10:   until no rules can be added to  $\Delta_d$ 
11:    $\mathcal{Q}_f \leftarrow \{Q' \in \mathcal{Q}_d \mid Q' \cap Q_f \neq \emptyset\}$ 
12:   return  $(\mathcal{Q}_d, \mathcal{Q}_f, \Sigma, \Delta_d)$ 
13: end procedure

```

Figure 2: Textbook Determinisation Algorithm

set of transitions  $\{f(s_1, \dots, s_n) \rightarrow q\} \cup \{q' \rightarrow s_i \mid q' \in Q_i, 1 \leq i \leq n\}$ . It can be shown that this transformation preserves the language of the FTA.

**Example 2.** *The transitions of the DFTA generated in Example 1 can be represented in product transition form as follows.*

$$\Delta' = \begin{array}{ll} \{\emptyset \rightarrow q_1 & 0 \rightarrow q_1 \\ [\{q_1, q_2\} \mid \{q_1\}] \rightarrow q_1 & [\{q_3\} \mid \{q_1\}] \rightarrow q_2 \\ [\{q_1, q_2\} \mid \{q_3\}] \rightarrow q_3 & [\{q_3\} \mid \{q_3\}] \rightarrow q_3 \\ [\{q_1, q_2\} \mid \{q_2\}] \rightarrow q_2 & [\{q_3\} \mid \{q_2\}] \rightarrow q_2 \end{array}$$

These 8 product transitions represent the 11 transitions shown in Example 1. There are more compact equivalent sets of product transitions, for example.

$$\Delta'' = \begin{array}{ll} \{\emptyset \rightarrow q_1 & 0 \rightarrow q_1 \\ [\{q_1, q_2, q_3\} \mid \{q_3\}] \rightarrow q_3 & [\{q_1, q_2, q_3\} \mid \{q_2\}] \rightarrow q_2 \\ [\{q_1, q_2\} \mid \{q_1\}] \rightarrow q_1 & [\{q_3\} \mid \{q_1\}] \rightarrow q_2 \end{array}$$

□

In the determinisation algorithm to be developed in the next section, product transitions for the output DFTA are generated directly; this leads in many cases to an exponential saving of space and time, as shown by the experiments in Section 6.

### 3. Development of an Optimised Determinisation Algorithm

In this section we present the textbook algorithm for FTA determinisation, and then proceed to optimise it. The starting point is the determinisation

algorithm in Figure 2, which is taken (apart from some renaming of variables) from [1].

The main idea of the transformation is to delay the computation of transitions  $\Delta_d$  until after the complete set of states of the DFTA,  $Q_d$ , has been computed. This has the following advantages.

- It allows a suitable data structure to be built, from which the set of transitions can be generated in product form.
- It enables significant optimisation of the main **repeat** loop to avoid redundant computations.
- It allows the computation of the set of transitions to be omitted entirely if desired, thus permitting other applications of the algorithm such as inclusion checking, which only requires the states.

We note first a small ambiguity in the algorithm as presented in [1]. In the assignment  $Q_0 = \{q_0 \mid \exists q_1 \in Q_1, \dots, q_n \in Q_n, (f(q_1, \dots, q_n) \rightarrow q_0) \in \Delta\}$  the right hand side is implicitly assumed to evaluate to a non-empty set, otherwise it is ignored. Although allowing the variable  $Q_0$  to take the value  $\emptyset$  would return a correct result, many redundant transitions of the form  $f(Q_1, \dots, Q_n) \rightarrow \emptyset$  would be generated. In our transformed algorithm we make this assumption explicit and eliminate such transitions.

Note that the states of the computed DFTA are elements of  $2^Q$  where  $Q$  is the set of states of the input FTA.

### 3.1. Introduction of functional notation

Let  $\langle Q, \Sigma, Q_f, \Delta \rangle$  be an FTA. Let  $\delta = f(q_1, \dots, q_n) \rightarrow q, n \geq 0$  be a transition in  $\Delta$ . Define the following selector functions on  $\delta$ .

$$\begin{array}{lll} \text{rhs} : \Delta \rightarrow Q & \text{lhs}_i : \Delta \hookrightarrow Q & \text{func} : \Delta \rightarrow Q \\ \text{rhs}(\delta) = q & \text{lhs}_i(\delta) = q_i, 1 \leq i \leq n & \text{func}(\delta) = f \end{array}$$

The  $\text{lhs}_i$  functions are partial functions on  $\Delta$  since  $\text{lhs}_i$  is not defined for every transition for a given  $i$ . In particular the  $\text{lhs}_i$  functions are undefined on transitions whose function symbol has arity zero.

The inverse mappings  $\text{lhs}_i^{-1} : Q \rightarrow 2^\Delta$  and  $\text{func}^{-1} : \Sigma \rightarrow 2^\Delta$  are defined respectively as  $\text{lhs}_i^{-1}(q) = \{\delta \mid \text{lhs}_i(\delta) = q\}$ ,  $\text{func}^{-1}(f) = \{\delta \mid \text{func}(\delta) = f\}$ . Using these,  $\text{lhsf}_i : (\Sigma \times Q) \rightarrow 2^\Delta$  is defined as  $\text{lhsf}_i(f, q) = \text{lhs}_i^{-1}(q) \cap \text{func}^{-1}(f)$ .

$\text{lhsf}_i(f, q)$  can be regarded as an index for  $\Delta$  returning the set of transitions whose function symbol is  $f$  and whose left hand side has  $q$  in the  $i^{\text{th}}$  position. The mappings  $\text{lhsf}_i$  are lifted to sets of states, giving  $\text{Lhsf}_i$  defined as follows.

$$\begin{array}{l} \text{Lhsf}_i : (\Sigma \times 2^Q) \rightarrow 2^\Delta \\ \text{Lhsf}_i(f, S) = \bigcup_{s \in S} \text{lhsf}_i(f, s) \end{array}$$

We also lift  $\text{rhs}$  to sets of transitions, giving the function  $\text{Rhs} : 2^\Delta \rightarrow 2^Q$ , where  $\text{Rhs}(T) = \{\text{rhs}(\delta) \mid \delta \in T\}$ .

The following property allows us to reformulate the algorithm, obtaining a form that is easier to manipulate.

**Lemma 1.** *The following expressions are equal for all  $f \in \Sigma$  and  $Q_1, \dots, Q_n \in 2^Q$ .*

$$\{q_0 \mid \exists q_1 \in Q_1, \dots, q_n \in Q_n, (f(q_1, \dots, q_n) \rightarrow q_0) \in \Delta\} \quad (1)$$

$$\begin{aligned} &\text{if } \text{ar}(f) = 0 \text{ then } \text{Rhs}(\text{func}^{-1}(f)) \text{ else} \\ &\quad \text{Rhs}(\text{Lhsf}_1(f, Q_1) \cap \dots \cap \text{Lhsf}_{\text{ar}(f)}(f, Q_{\text{ar}(f)})) \end{aligned} \quad (2)$$

**Proof 1.** *Application of the definitions of  $\text{Rhs}$ ,  $\text{Lhsf}_i$  and set operations.*  $\square$

### 3.2. Delaying computation of transitions

Examining the **repeat** loop in Figure 2, we observe that the values of  $\Delta_d$  and  $\mathcal{Q}_d$  are initialised to  $\emptyset$  before the first iteration of the loop and recomputed on each iteration of the loop body. Let us represent the action of the loop body as a function  $\phi$ , and express the effect of the loop body as  $(\mathcal{Q}'_d, \Delta'_d) \leftarrow \phi(\mathcal{Q}_d, \Delta_d)$ .

We can then summarise the execution of the **repeat** loop as the sequence

$$\begin{aligned} &(\mathcal{Q}_d^1, \Delta_d^1) \leftarrow \phi(\emptyset, \emptyset) \\ &(\mathcal{Q}_d^2, \Delta_d^2) \leftarrow \phi(\mathcal{Q}_d^1, \Delta_d^1) \\ &\dots \\ &(\mathcal{Q}_d^{k-1}, \Delta_d^{k-1}) \leftarrow \phi(\mathcal{Q}_d^{k-2}, \Delta_d^{k-2}) \\ &(\mathcal{Q}_d^k, \Delta_d^k) \leftarrow \phi(\mathcal{Q}_d^{k-1}, \Delta_d^{k-1}) \end{aligned}$$

and the loop terminates when  $\Delta_d^k = \Delta_d^{k-1}$ . We can establish that the above iteration sequence could be replaced by the following one, in which the value of  $\Delta_d$  does not accumulate but is computed “from scratch” on each iteration, in effect executing  $\Delta_d \leftarrow \emptyset$  at the start of each iteration. This is because the set of transitions derived in each iteration depends only on the value of  $\mathcal{Q}_d$  at the start of the loop body, and not on the current value of  $\Delta_d$ .

$$\begin{aligned} &(\mathcal{Q}_d^1, \Delta_d^1) \leftarrow \phi(\emptyset, \emptyset) \\ &(\mathcal{Q}_d^2, \Delta_d^2) \leftarrow \phi(\mathcal{Q}_d^1, \emptyset) \\ &\dots \\ &(\mathcal{Q}_d^{k-1}, \Delta_d^{k-1}) \leftarrow \phi(\mathcal{Q}_d^{k-2}, \emptyset) \\ &(\mathcal{Q}_d^k, \Delta_d^k) \leftarrow \phi(\mathcal{Q}_d^{k-1}, \emptyset) \end{aligned}$$

The sequence of values  $(\mathcal{Q}_d^1, \Delta_d^1), \dots, (\mathcal{Q}_d^k, \Delta_d^k)$  is the same in the two iteration sequences, except that possibly the second sequence is shorter by one, since the termination condition for the **repeat** loop is then changed to  $\mathcal{Q}_d^k = \mathcal{Q}_d^{k-1}$ , possibly reducing the number of iterations by one, as the value  $\mathcal{Q}_d$  can stabilise one iteration earlier than the value of  $\Delta_d$  does. Comparison of the sequences shows that the computation of  $\Delta_d$  can be removed from the **repeat** loop entirely, delaying it until after the termination of the loop. From the second sequence we can see that the final value of  $\Delta_d$  can be computed from the final value of  $\mathcal{Q}_d$  with one extra execution of the loop body. We return to the computation of  $\Delta_d$  in Section 3.5.

The next stage of the transformed algorithm after applying these transformations is displayed in Figure 3, where the processing of 0-arity functions, which

```

1: procedure FTA DETERMINISATION (Input:  $\langle Q, \Sigma, Q_f, \Delta \rangle$ )
2:    $\mathcal{Q}_d \leftarrow \emptyset$ 
3:   for all  $f \in \Sigma$  do
4:     if  $(\text{ar}(f) = 0)$  then
5:        $Q_0 \leftarrow \text{Rhs}(\text{func}^{-1}(f))$ 
6:       if  $Q_0 \neq \emptyset$  then
7:          $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$ 
8:       end if
9:     end if
10:  end for
11:  repeat
12:     $\mathcal{Q}_d^{old} \leftarrow \mathcal{Q}_d$ 
13:    for all  $f \in \Sigma$  do
14:      if  $(\text{ar}(f) > 0)$  then
15:         $n \leftarrow \text{ar}(f)$ 
16:        for all  $(Q_1, \dots, Q_n) \in (\mathcal{Q}_d^{old} \times \dots \times \mathcal{Q}_d^{old})$  do
17:           $Q_0 \leftarrow \text{Rhs}(\text{Lhsf}_1(f, Q_1) \cap \dots \cap \text{Lhsf}_n(f, Q_n))$ 
18:          if  $Q_0 \neq \emptyset$  then
19:             $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$ 
20:          end if
21:        end for
22:      end if
23:    end for
24:  until  $\mathcal{Q}_d = \mathcal{Q}_d^{old}$ 
25:  Compute the set of transitions  $\Delta_d$  (see Section 3.5)
26:   $\mathcal{Q}_f \leftarrow \{Q' \in \mathcal{Q}_d \mid Q' \cap Q_f \neq \emptyset\}$ 
27:  return  $(\mathcal{Q}_d, \Sigma, \mathcal{Q}_f, \Delta_d)$ 
28: end procedure

```

Figure 3: Algorithm after delaying the computation of transitions

depends only on the original FTA, is completed (lines 3-10) before entering the main loop.

The transformations so far are fairly superficial and have little bearing on the efficiency of the algorithm. However they enable us to focus on the iterations of the inner **for all** loop, with a view to more substantial efficiency improvements.

### 3.3. Inner Loop Optimisation

The fact that we no longer need to compute transitions in the inner loop can lead to major savings since we can focus on the computation of  $\mathcal{Q}_d$ . Let us suppose that  $|\mathcal{Q}_d^{old}| = k$  in Figure 3 (line 12). Then for a function symbol  $f$  of arity  $n$ , there are  $k^n$  tuples  $(Q_1, \dots, Q_n)$  in the cartesian product  $(\mathcal{Q}_d^{old} \times \dots \times \mathcal{Q}_d^{old})$  and so the function  $\text{Lhsf}_i(f, Q_j)$  is called  $n * k^n$  times. On the other hand, within the loop there are only  $k * n$  different calls of the form  $\text{Lhsf}_i(f, Q_j)$  and therefore it is worth precomputing these  $k * n$  values outside the loop and



avoid recomputing the same call many times. Furthermore, cases of  $\text{Lhsf}_i(f, Q_j)$  that evaluate to the empty set can be ignored since they cannot contribute to a non-empty value of  $Q_0$  within the loop, since  $\text{Rhs}(\emptyset) = \emptyset$ .

We precompute the  $\text{Lhsf}_i(f, Q_j)$  values by introducing a function called  $\mathcal{T}_i : (\Sigma \times 2^{2^Q}) \rightarrow 2^{2^\Delta}$  defined as

$$\mathcal{T}_i(f, \mathcal{Q}') = \{\text{Lhsf}_i(f, Q') \mid Q' \in \mathcal{Q}'\} \setminus \{\emptyset\}.$$

This function is defined for  $1 \leq i \leq n$  for a function of arity  $n$  and returns a set of sets of transitions. The inner **for all** loop is then rewritten to iterate over tuples of sets of transitions chosen from the product  $\mathcal{T}_1(f, \mathcal{Q}_d^{old}) \times \dots \times \mathcal{T}_n(f, \mathcal{Q}_d^{old})$  instead of  $(\mathcal{Q}_d^{old} \times \dots \times \mathcal{Q}_d^{old})$ .

It can be seen that the same non-empty values of  $Q_0$  are generated within the loop. Whereas previously we iterated over tuples  $(Q_1, \dots, Q_n)$  in the cartesian product  $(\mathcal{Q}_d^{old} \times \dots \times \mathcal{Q}_d^{old})$ , and then applied  $\text{Rhs}(\text{Lhsf}_1(f, Q_1) \cap \dots \cap \text{Lhsf}_n(f, Q_n))$ , now we precompute all the possible  $\text{Lhsf}_i(f, Q_i)$  values (which are sets of transitions  $\Delta_i$ ) and then call  $\text{Rhs}(\Delta_1 \cap \dots \cap \Delta_n)$  for every possible tuple of non-empty sets of transitions  $(\Delta_1, \dots, \Delta_n)$ .

The transformation of the inner loop is significant in typical applications. Instead of  $k^n$  iterations of the loop, where  $k = |\mathcal{Q}_d^{old}|$ , there are  $\prod_{i=1}^n |\mathcal{T}_i(f, \mathcal{Q}_d^{old})|$  iterations which is usually much smaller. Note that in many FTAs the size of the set  $\mathcal{T}_i(f, \mathcal{Q}_d^{old})$  is much smaller than  $k$  (and is often zero) since the states of the input automaton tend to appear in only a few argument positions of function symbols.

#### 3.4. Tracking new values on each iteration

We now apply an optimisation that further reduces the computation in the innermost loop. As it stands in Figure 3, any state  $Q_0$  generated on some iteration at line 17 is also generated on all subsequent iterations of the **repeat** loop. To avoid this, we note that when evaluating the statement  $Q_0 \leftarrow \text{Rhs}(\Delta_1 \cap \dots \cap \Delta_n)$  in some iteration of the **repeat** loop, a new value is obtained for  $Q_0$  only when at least one of  $\Delta_1, \dots, \Delta_n$  is a new value, that is, one that was not available on the previous iteration. We therefore try to avoid re-evaluating old values of  $\Delta_i$  for each  $i$ .

Some bookkeeping is needed to keep track of new values. A variable  $\mathcal{Q}_d^{new}$  represents the new elements of  $\mathcal{Q}_d$  produced on some iteration. (The termination condition of the **repeat** loop is altered to  $\mathcal{Q}_d^{new} = \emptyset$ ). We introduce variables  $\Psi_i^f$ , which have the value of  $\mathcal{T}_i(f, \mathcal{Q}_d)$ . The variables are initialised to  $\emptyset$  and their values are augmented on each iteration. The statement  $(\Phi_1, \dots, \Phi_n) \leftarrow (\mathcal{T}_1(f, \mathcal{Q}_d^{new}) \setminus \Psi_1^f, \dots, \mathcal{T}_n(f, \mathcal{Q}_d^{new}) \setminus \Psi_n^f)$  computes the new sets of transitions from which the  $\Delta_i$  sets can be chosen.

The innermost **for all** statement now iterates over the union of sets of tuples  $Z$  where  $Z = \bigcup_{i=1}^n (\Psi_1^f \times \dots \times \Psi_{i-1}^f \times \Phi_i \times \Psi_{i+1}^{f, new} \times \dots \times \Psi_n^{f, new})$ , which consists of exactly those tuples which contain at least one new value (that is, from one of the  $\Phi_i$ ). Note in particular that if no new values for any argument are produced for some  $f$  on some iteration, then the iteration set for  $f$  on the next iteration is empty, since all the variables  $(\Phi_1, \dots, \Phi_n)$  have the value  $\emptyset$ .

```

1:  $\Delta_d \leftarrow \emptyset$ 
2: for all  $f \in \Sigma$  do
3:   if ( $\text{ar}(f) = 0$ ) then
4:      $Q_0 \leftarrow \text{Rhs}(\text{func}^{-1}(f))$ 
5:     if  $Q_0 \neq \emptyset$  then
6:        $\Delta_d \leftarrow \Delta_d \cup \{f \rightarrow Q_0\}$ 
7:     end if
8:   else
9:      $n \leftarrow \text{ar}(f)$ 
10:     $\Psi_1^f, \dots, \Psi_n^f \leftarrow \mathcal{T}_1(f, \mathcal{Q}_d), \dots, \mathcal{T}_n(f, \mathcal{Q}_d)$ 
11:    for all  $(\Delta_1, \dots, \Delta_n) \in (\Psi_1^f \times \dots \times \Psi_n^f)$  do
12:       $Q_0 \leftarrow \text{Rhs}(\Delta_1 \cap \dots \cap \Delta_n)$ 
13:      if  $Q_0 \neq \emptyset$  then
14:         $\mathcal{Q}_1, \dots, \mathcal{Q}_n \leftarrow \mathcal{T}_1^{-1}(f, \mathcal{Q}_d, \Delta_1), \dots, \mathcal{T}_n^{-1}(f, \mathcal{Q}_d, \Delta_n)$ 
15:         $\Delta_d \leftarrow \Delta_d \cup \{f(\mathcal{Q}_1, \dots, \mathcal{Q}_n) \rightarrow Q_0\}$ 
16:      end if
17:    end for
18:  end if
19: end for

```

Figure 4: Computation of product transitions

### 3.5. Computing the transitions of the DFTA in product form

As noted in Section 3.2, the set of transitions  $\Delta_d$  of the determinised automaton can be computed from the final set of states  $\mathcal{Q}_d$  with one extra iteration of the **repeat** loop.

The idea is that, at the point in Figure 3 where state  $Q_0$  is generated (line 17), we could have generated a transition  $f(Q_1, \dots, Q_n) \rightarrow Q_0$ . However, following the optimisations of the inner loop described in Section 3.3, we no longer compute the tuple  $(Q_1, \dots, Q_n)$  explicitly. Instead, we keep track of the information needed to generate the tuple  $(Q_1, \dots, Q_n)$  and the function  $f$ , so that we can construct the transition later.

To do this efficiently, we introduce another function  $\mathcal{T}_i^{-1} : (\Sigma \times 2^{2^Q} \times 2^\Delta) \rightarrow 2^{2^Q}$ , defined as follows.

$$\mathcal{T}_i^{-1}(f, \mathcal{Q}, \Delta') = \{Q' \mid \text{Lhs}f_i(f, Q') = \Delta', Q' \in \mathcal{Q}\}$$

The set of transitions for  $f$  is then all transitions of the form  $f(Q_1, \dots, Q_n) \rightarrow \text{Rhs}(\Delta_1 \cap \dots \cap \Delta_n)$  such that  $(\Delta_1, \dots, \Delta_n) \in (\mathcal{T}_1(f, \mathcal{Q}_d) \times \dots \times \mathcal{T}_n(f, \mathcal{Q}_d))$  and  $(Q_1, \dots, Q_n) \in (\mathcal{T}_1^{-1}(f, \mathcal{Q}_d, \Delta_1) \times \dots \times \mathcal{T}_n^{-1}(f, \mathcal{Q}_d, \Delta_n))$ .

Product transitions for  $\Delta_d$  can then be obtained directly. We simply omit the enumeration of the tuples  $Q_1, \dots, Q_n$  which are elements of  $\mathcal{T}_1^{-1}(f, \mathcal{Q}_d, \Delta_1) \times \dots \times \mathcal{T}_n^{-1}(f, \mathcal{Q}_d, \Delta_n)$ . This gives the algorithm in Figure 4 for generating  $\Delta_d$  in product form. In the expression  $f(\mathcal{Q}_1, \dots, \mathcal{Q}_n) \rightarrow Q_0$  in Figure 4,  $\mathcal{Q}_1, \dots, \mathcal{Q}_n$  are sets of DFTA states.

```

1: procedure FTA DETERMINISATION (Input:  $\langle Q, \Sigma, Q_f, \Delta \rangle$ )
2:    $\mathcal{Q}_d \leftarrow \emptyset$ 
3:   for all  $f \in \Sigma$  do
4:     if  $\text{ar}(f) = 0$  then
5:        $Q_0 \leftarrow \text{Rhs}(\text{func}^{-1}(f))$ 
6:       if  $Q_0 \neq \emptyset$  then
7:          $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$ 
8:       end if
9:     end if
10:     $(\Psi_1^f, \dots, \Psi_n^f) \leftarrow (\emptyset, \dots, \emptyset)$ 
11:  end for
12:   $\mathcal{Q}_d^{new} \leftarrow \mathcal{Q}_d$ 
13:  repeat
14:     $\mathcal{Q}_d^{old} \leftarrow \mathcal{Q}_d$ 
15:    for all  $f \in \Sigma$  do
16:      if  $\text{ar}(f) > 0$  then
17:         $n \leftarrow \text{ar}(f)$ 
18:         $(\Phi_1, \dots, \Phi_n) \leftarrow (\mathcal{T}_1(f, \mathcal{Q}_d^{new}) \setminus \Psi_1^f, \dots, \mathcal{T}_n(f, \mathcal{Q}_d^{new}) \setminus \Psi_n^f)$ 
19:         $(\Psi_1^{f,new}, \dots, \Psi_n^{f,new}) \leftarrow (\Psi_1^f \cup \Phi_1, \dots, \Psi_n^f \cup \Phi_n)$ 
20:         $Z \leftarrow \bigcup_{i=1}^n (\Psi_1^f \times \dots \times \Psi_{i-1}^f \times \Phi_i \times \Psi_{i+1}^{f,new} \times \dots \times \Psi_n^{f,new})$ 
21:        for all  $(\Delta_1, \dots, \Delta_n) \in Z$  do
22:           $Q_0 \leftarrow \text{Rhs}(\Delta_1 \cap \dots \cap \Delta_n)$ 
23:          if  $Q_0 \neq \emptyset$  then
24:             $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$ 
25:          end if
26:        end for
27:      end if
28:       $(\Psi_1^f, \dots, \Psi_n^f) \leftarrow (\Psi_1^{f,new}, \dots, \Psi_n^{f,new})$ 
29:    end for
30:     $\mathcal{Q}_d^{new} \leftarrow \mathcal{Q}_d \setminus \mathcal{Q}_d^{old}$ 
31:  until  $\mathcal{Q}_d^{new} = \emptyset$ 
32:  Compute the set of transitions  $\Delta_d$  (see Figure 4)
33:   $\mathcal{Q}_f \leftarrow \{Q' \in \mathcal{Q}_d \mid Q' \cap Q_f \neq \emptyset\}$ 
34:  return  $(\mathcal{Q}_d, \Sigma, \mathcal{Q}_f, \Delta_d)$ 
35: end procedure

```

Figure 5: Optimised algorithm

### 3.6. Further optimisation

The final algorithm then consists of Figure 5 for computing  $\mathcal{Q}_d$  together with Figure 4 for computing  $\Delta_d$  in product form. We note that in Figure 4 for computing  $\Delta_d$ , the values of  $\Psi_1^f, \dots, \Psi_n^f$  are available from the main **repeat** loop in Figure 5 and do not need to be recomputed. Also, the values of the expressions  $\mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta_i)$  can be computed in the main loop of Figure 5 by tabulating computed values of  $\text{Lhsf}_i$ ; more precisely, whenever an expression  $\text{Lhsf}_i(f, Q')$  is evaluated and yields a non-empty value  $\Delta'$ ,  $Q'$  is added to the set of values  $\mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$ .

## 4. Performance of the optimised algorithm

*Worst Case.* Consider first the worst case running time for determinising the FTA  $\langle Q, F, \Sigma, \Delta \rangle$ . The size of the input is measured by  $|\Sigma|$ ,  $|Q|$  and  $n$ , the maximum arity of the elements of  $\Sigma$ .

The main **repeat** loop of the algorithm in Figure 5 can be traversed up to  $2^{|Q|}$  times, which is the upper bound of the number of states in the DFTA. For each  $f \in \Sigma$  within the main loop there are up to  $(2^{|Q|})^n$  iterations, since the size of  $\mathcal{T}_i(f, \mathcal{Q}_d^{old})$  can be up to  $|\mathcal{Q}_d| = 2^{|Q|}$ . Combining all three nested loops, the complexity of the main loop of the algorithm is  $O(|\Sigma| \cdot 2^{|Q| \cdot (n+1)})$ .

Considering the product transitions, the maximum number of iterations of the transition-generation loop for each  $n$ -ary  $f \in \Sigma$  is  $(2^{|Q|})^n$  and there is at most one product transition generated in each iteration. Hence the number of product transitions generated in the worst case is  $O(|\Sigma| \cdot 2^{n \cdot |Q|})$ . If the automaton is also completed, as discussed in Section 5, then  $|Q|$  is possibly increased by one (the state *any*).

Regarding both the running time and the size of the output, the optimised algorithm performs no better in the worst case than the textbook algorithm with explicit enumeration of the DFTA transitions.

*Running time in practice.* The worst case of  $2^{|Q|}$  for the number of DFTA states seems to be approached only in unusual situations; to achieve it, for every pairs of states  $q, q'$  of the original automaton it would have to be the case that  $L(q) \cap L(q')$ ,  $L(q) \setminus L(q')$  and  $L(q') \setminus L(q)$  are all nonempty (since the states of the generated DFTA include only ones accepting some term, see Lemma 2). For such a pair of states, it is more common either that  $L(q)$  and  $L(q')$  are disjoint or that one includes the other. If this is the case for all such pairs, then the number of DFTA states is at most the number of original states. In Example 1 the number of DFTA states is the same as the number of original states; our experiments (Section 6) show a tendency for the size of the set of states of the FTA and the corresponding DFTA to be close for examples where the FTA is a description of program data. The set of DFTA states may even be smaller, if the original FTA has redundant or duplicated states - which sometimes happens with automatically generated FTAs.

Replacing  $2^{|Q|}$  by  $|\mathcal{Q}_d|$  in the complexity expressions gives a running time of  $O(|\Sigma| \cdot |\mathcal{Q}_d|^{n+1})$  and  $O(|\Sigma| \cdot |\mathcal{Q}_d|^n)$  for the number of product transitions in

the output. Even if  $|Q_d|$  is small, we can see that high-arity function symbols still present a potential for blow-up. Again, in practice this danger is often greatly reduced by the structure of the input FTA. As already noted, the size of  $\mathcal{T}_i(f, Q_d)$ , whose worst-case size is  $|Q_d|$  is usually much smaller than  $|Q_d|$ . This is due to the natural “typing” of function symbols. A function argument position in the original FTA is typically associated with a small number of states. However, there are applications where the size of  $\mathcal{T}_i(f, Q_d)$  is larger and for these is a danger of blow-up for high-arity function symbols. Such applications will be discussed in Section 6.

## 5. Complete DFTAs

Recall that in a complete FTA (Definition 2) every term  $t \in \text{Term}(\Sigma)$  has a run  $t \Rightarrow^* q$  where  $q \in Q$ . An FTA can always be completed [1], by adding an extra state to  $Q$  and adding extra transitions to  $\Delta$ .

**Example 3.** Consider the following FTA  $\langle Q, \Sigma, Q_f, \Delta \rangle$  where  $Q = \{\text{list}, \text{num}\}$ ,  $Q_f = \{\text{list}\}$ ,  $\Sigma = \{\square, [\cdot|\cdot], 0, s(\cdot)\}$ , and  $\Delta = \{\square \rightarrow \text{list}, [\text{num}|\text{list}] \rightarrow \text{list}, 0 \rightarrow \text{num}, s(\text{num}) \rightarrow \text{num}\}$ . The FTA is not complete; for instance there is no transition with left hand side  $s(\text{list})$  or  $[\text{num}|\text{num}]$ . Thus the terms  $s(\square)$  and  $[s(0)|0]$ , for example, have no run. To complete it, we can add an extra non-final state, say  $e$  (for error), and add the following transitions.

$$\Delta_e = \begin{aligned} & \{s(\text{list}) \rightarrow e, s(e) \rightarrow e, [\text{list}|\text{list}] \rightarrow e, \\ & [\text{num}|\text{num}] \rightarrow e, [\text{list}|\text{num}] \rightarrow e, [e|\text{list}] \rightarrow e, \\ & [\text{list}|e] \rightarrow e, [e|e] \rightarrow e, [\text{num}|e] \rightarrow e, [e|\text{num}] \rightarrow e\} \end{aligned}$$

The FTA  $\langle Q \cup \{e\}, \Sigma, Q_f, \Delta \cup \Delta_e \rangle$  is complete and accepts the same language as the original FTA (the set of lists of numbers). Any term that did not have a run in the original now has a run to  $e$  (which is not an accepting state).

One can see that the number of transitions in the completed FTA is determined by the arity of the function symbols and the number of states, and that it is exponential in the arity of the function symbols.

It would be possible to modify the textbook determinisation procedure in Figure 2 to complete the FTA as well as determinise it. However, it would then be more involved to apply the optimisations and generate product transitions. The approach given in the next section is to add transitions to the input FTA (namely  $\Delta_{\text{any}}$ ) that guarantee that the resulting DFTA is complete. The optimisations introduced in the previous sections then take effect, and transitions are generated in product form, without any further modification of the algorithm.

### 5.1. Simultaneous completion and determinisation using $\Delta_{\text{any}}$

Recall that given a finite signature  $\Sigma$  we define the set of transitions  $\Delta_{\text{any}}$  as  $\{f(\overbrace{\text{any}, \dots, \text{any}}^{n \text{ times}}) \rightarrow \text{any} \mid f^n \in \Sigma\}$ . Clearly for any term  $t \in \text{Term}(\Sigma)$  there exists a run  $t \Rightarrow^* \text{any}$ . The following lemma shows that we can use  $\Delta_{\text{any}}$  to

obtain a complete DFTA from a given FTA, in other words, we can perform determinisation and completion simultaneously. First we establish an important property of the DFTAs generated by the algorithm - that they contain only “productive” states.

**Lemma 2.** *Given an FTA, for every state  $Q'$  of the DFTA obtained by determinising it using the given (optimised) algorithm,  $L(Q') \neq \emptyset$ .*

**Proof 2.** *We consider the algorithm in Figure 3 for simplicity, rather than the final optimised algorithm. Consider a DFTA state  $Q'$  that is generated in the algorithm. We reason by induction on the number of iterations of the **repeat** loop of the algorithm.*

- *Base case ( $i = 0$ ): If  $Q'$  is generated before the first iteration, then there exists some 0-arity function  $f$  such that  $Q' = \text{Rhs}(\text{func}^{-1}(f))$ . The transition  $f \rightarrow Q'$  is generated on iteration 1 hence there is a run  $f \Rightarrow Q'$ .*
- *Induction ( $i > 1$ ): Assume that the lemma holds for all states generated up to the  $i - 1^{\text{th}}$  iteration, and that  $Q'$  is a new state generated on the  $i^{\text{th}}$  iteration. Then there exist states  $Q_1, \dots, Q_n$  which were generated in the first  $i - 1$  iterations, such that a transition  $f(Q_1, \dots, Q_n) \rightarrow Q'$  is constructed on the  $i^{\text{th}}$  iteration. By the inductive hypothesis, there exist terms  $t_1, \dots, t_n$  such that  $t_i \Rightarrow^* Q_i, 1 \leq i \leq n$ . Hence there is a run  $f(t_1, \dots, t_n) \Rightarrow^* Q'$ .*

Hence the lemma holds for states generated on any iteration.  $\square$

**Lemma 3.** *Let  $\langle Q, Q_f, \Sigma, \Delta \rangle$  be an FTA such that every term  $t \in \text{Term}(\Sigma)$  has a run  $t \Rightarrow^* q$  for some  $q \in Q$ . Then the DFTA obtained by determinising  $\langle Q, Q_f, \Sigma, \Delta \rangle$  is complete.*

**Proof 3.** *Let  $\mathcal{Q}'$  be the set of states of the generated DFTA, and assume that it is not complete. Then there exist  $Q_1, \dots, Q_n \in \mathcal{Q}'$  and an  $n$ -ary function  $f$  such that for all  $Q_0$  there is no transition  $f(Q_1, \dots, Q_n) \rightarrow Q_0$  in the DFTA. Let  $t_1, \dots, t_n \in \text{Term}(\Sigma)$  be terms such that  $t_i \Rightarrow^* Q_i, 1 \leq i \leq n$ , whose existence is guaranteed by Lemma 2. Since it is a DFTA, there cannot be any other runs  $t_i \Rightarrow^* Q'_i$  where  $Q_i \neq Q'_i$  respectively. Hence there is no run for the term  $f(t_1, \dots, t_n)$ , which contradicts the assumption of the lemma. Hence the generated DFTA is complete.  $\square$*

Thus we establish the following.

**Proposition 1.** *Let  $\langle Q, Q_f, \Sigma, \Delta \rangle$  be an FTA, such that any  $\in Q$  and  $\Delta_{\text{any}} \subseteq \Delta$ . Then the DFTA obtained by the determinisation algorithm with this input is complete.*

**Proof 4.** *For every term  $t \in \text{Term}(\Sigma)$  the FTA has a run  $t \Rightarrow^* \text{any}$ . The result follows from Lemma 3.  $\square$*

### 5.2. Performance of the algorithm after adding $\Delta_{any}$

Although by Proposition 1 we can obtain a complete DFTA from any given input FTA, simply by adding the state *any* and the transitions  $\Delta_{any}$  to the input before running the algorithm, we may ask what is the impact on the performance of the determinisation algorithm.

The impact on the number of states of the DFTA is slight; at most one extra state  $\{any\}$  is generated, in the case that there are some terms accepted by *any* but not by any other state. This state represents the “error” state of the classical completion procedure. Apart from this, the same states are generated but *any* is added to each one; it is easy to see that *any* must appear in every DFTA state since the state *any* appears in every possible left-hand-side position in  $\Delta_{any}$ .

With standard transitions, completion can cause a huge increase in the size of  $\Delta_d$ . The main question is thus the impact on the product representation of  $\Delta_d$ . Let us analyse the effect of adding  $\Delta_{any}$  on part of the algorithm generating the transitions of the DFTA in product form (Figure 4). Consider the effect of the introduction of  $\Delta_{any}$  on the basic operations of the algorithm.

- $\text{func}^{-1}(f)$  and  $\text{Lhsf}_i(f, Q)$ : in each case the returned set contains at most one extra transition (that is, the transition  $f(any, \dots, any) \rightarrow any$ , in the case that  $any \in Q$ ).
- $\text{Rhs}(T)$ : the returned set contains at most one extra state *any* in the case that  $T \cap \Delta_{any} \neq \emptyset$ .

The worst case of the number of extra iterations of the inner loop in Figure 4 caused by the extra state is discussed in Section 4.

In Section 6 we verify that the time overhead of adding  $\Delta_{any}$  is small. The overhead in the number of product transition is discussed in Section 6, and is in practice a small factor. Thus we obtain the completed DFTA at little extra cost over obtaining the DFTA.

### 5.3. Don't-Care Arguments in Complete DFTAs

An underscore “ $\_$ ” is used as shorthand for  $\mathcal{Q}_d$  in a product transition; that is,  $f(\mathcal{Q}_1, \dots, \_, \dots, \mathcal{Q}_n) \rightarrow Q_0$  is the same as  $f(\mathcal{Q}_1, \dots, \mathcal{Q}_d, \dots, \mathcal{Q}_n) \rightarrow Q_0$ . This indicates that the choice of DFTA state in this argument is irrelevant in determining the right hand side  $Q_0$ . Product transitions of the form  $f(\_, \dots, \mathcal{Q}_i, \dots, \_) \rightarrow Q_0$  in which all but one argument are don't-care arguments are especially interesting, since the right-hand-side of the transition is determined by just one argument. We call the elements of such arguments *deciding arguments*.

A typical case of deciding arguments arises in complete DFTAs constructed by adding  $\Delta_{any}$  to the original FTA, where a state  $\{any\}$  is generated, which accepts the terms not accepted by any other state.  $\{any\}$  is a deciding argument; the presence of  $\{any\}$  in any argument in a DFTA transition is sufficient to

ensure that the right hand side of the transition is  $\{any\}$ . That is, there are DFTA product transitions of the form:

$$\begin{aligned} f(\{\{any\}\}, \_, \dots, \_, \_) &\rightarrow \{any\} \\ f(\_, \{\{any\}\}, \_, \dots, \_) &\rightarrow \{any\} \\ &\vdots \\ f(\_, \_, \dots, \_, \{\{any\}\}) &\rightarrow \{any\} \end{aligned}$$

These product transitions overlap, obviously, since  $\{any\}$  is included in the don't-care arguments. However, this form might be much more compact than the product transitions generated by the determinisation algorithm. Furthermore, there could be other deciding arguments besides  $\{any\}$ .

We prove two lemmas defining sufficient conditions for finding deciding arguments and generating the corresponding don't-care product transitions.

**Lemma 4.** *Let  $\mathcal{Q}_d$  be the set of states of a complete DFTA and let  $\Psi_1, \dots, \Psi_n = \mathcal{T}_1(f, \mathcal{Q}_d), \dots, \mathcal{T}_n(f, \mathcal{Q}_d)$  for some  $n$ -ary function  $f$ . Let  $\Delta' \in \Psi_i$  and  $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$ . Then  $\mathcal{Q}_i$  are deciding arguments for the  $i^{th}$  argument of  $f$  if*

$$\text{Rhs}(\Delta' \cap \bigcap (\cap \Psi_1, \dots, \cap \Psi_{i-1}, \cap \Psi_{i+1}, \dots, \cap \Psi_n)) = \text{Rhs}(\Delta').$$

**Proof 5.** *Consider the set of right hand sides of transitions that can be built from  $(\Psi_1 \times \dots \times \Psi_n)$  using  $\Delta'$  in the  $i^{th}$  position, say  $\bar{Q}$ . That is,  $\bar{Q} = \text{Rhs}(\Delta_1 \cap \dots \cap \Delta' \cap \dots \cap \Delta_n)$  where  $(\Delta_1, \dots, \Delta', \dots, \Delta_n) \in (\Psi_1 \times \dots \times \{\Delta'\} \times \dots \times \Psi_n)$  with  $\Delta'$  in the  $i^{th}$  position.*

*$\text{Rhs}(\Delta')$  is an upper bound for  $\bar{Q}$ , since  $\Delta_1 \cap \dots \cap \Delta' \cap \dots \cap \Delta_n \subseteq \Delta'$  and  $\text{Rhs}$  is monotonic. The expression  $\text{Rhs}(\Delta' \cap \bigcap (\cap \Psi_1, \dots, \cap \Psi_{i-1}, \cap \Psi_{i+1}, \dots, \cap \Psi_n))$  is a lower bound for  $\bar{Q}$ , since  $\Delta' \cap \bigcap (\cap \Psi_1, \dots, \cap \Psi_{i-1}, \cap \Psi_{i+1}, \dots, \cap \Psi_n) \subseteq \Delta_1 \cap \dots \cap \Delta' \cap \dots \cap \Delta_n$  and  $\text{Rhs}$  is monotonic. If these two are equal, as in the statement of the property, then we can conclude that the value  $\text{Rhs}(\Delta')$  is the right hand side for any such transition since it is both an upper and a lower bound.*

□

If we can find such a  $\Delta' \in \mathcal{T}_i(f, \mathcal{Q}_d)$ , a (product) transition  $f(\dots, \_, \mathcal{Q}_i, \_, \dots) \rightarrow \text{Rhs}(\Delta')$  is constructed, where  $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$  and the underscore arguments stand for  $\mathcal{Q}_d$ .

A more specialised sufficient condition for deciding arguments for binary functions is given by the following lemma.

**Lemma 5.** *Let  $\mathcal{Q}_d$  be the set of states of a complete DFTA and let  $\Psi_1, \Psi_2 = \mathcal{T}_1(f, \mathcal{Q}_d), \mathcal{T}_2(f, \mathcal{Q}_d)$  for some 2-ary function  $f$ . Then a set of DFTA states  $\mathcal{Q}_i \subseteq \mathcal{Q}_d$ , where  $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$  for some  $\Delta' \in \Psi_i$ ,  $i \in \{1, 2\}$  are deciding arguments for the  $i^{th}$  argument of  $f$  if  $\text{Rhs}(\Delta')$  is a singleton, and for all  $\Delta'' \in \Psi_j$ ,  $j \in \{1, 2\} \setminus \{i\}$ ,*

$$\Delta' \cap \Delta'' \neq \emptyset.$$



**Proof 6.**  $\Delta' \cap \Delta'' \neq \emptyset$  implies that  $\text{Rhs}(\Delta_j \cap \Delta') \neq \emptyset$ . Since  $\text{Rhs}(\Delta')$  is a singleton and  $\text{Rhs}(\Delta'' \cap \Delta') \subseteq \text{Rhs}(\Delta')$ ,  $\text{Rhs}(\Delta'' \cap \Delta') = \text{Rhs}(\Delta')$ .

If such a  $\Delta'$  is found, say in argument 2, we generate the product transition  $f(\_, Q_2) \rightarrow \text{Rhs}(\Delta')$  where  $Q_2 = \mathcal{T}_2^{-1}(f, Q_d, \Delta')$  and the underscore arguments stand for  $Q_d$ .

We can easily add a check for deciding arguments using the conditions of Lemma 4 and 5 to the algorithm, just before generating product transitions. The calculation of the intersections is exponential in the arity of the function symbols, but does not alter the complexity of the overall algorithm and can save effort in generating product transitions. For each set of deciding arguments  $Q_i$  discovered, we generate a don't-care product transition of the form just shown, and the corresponding value  $\Delta'$  is removed from  $\Psi_i$  when computing the remaining product transitions for  $f$ .

The problem of finding the minimum number of product transitions to represent the DFTA transitions seems to be intractable and is beyond the scope of this paper. In essence it can be stated as the problem of finding the minimum number of cartesian products whose union is a given relation.

## 6. Experiments

Tables 1 and 2 show experimental results comparing the optimised determinisation algorithms generating transitions in product form with the textbook algorithm. It also compares the effect of adding the detection of don't care arguments in the determinisation algorithm. The algorithms are implemented in Java;<sup>2</sup> the textbook algorithm is a fairly direct implementation of the program in Figure 3. Our own implementation of the textbook algorithm could be improved but the same overall performance is likely in terms of the number of solvable problems, since the size of the output set of transitions is often the limiting factor. In Section 6.2, we compare with other implementations of FTA operations, where we find that our implementation of the textbook algorithm seems comparable in performance with existing determinisation tools.

The 14,694 benchmark FTAs were obtained from the repository that is part of libvata. Many of these FTAs originate in the Timbuk system [2] and others from the abstract regular tree model checking experiments (ARTMC) [3]. All experiments were carried out on a computer running Linux with 4-core Intel<sup>®</sup> Xeon<sup>®</sup> X5355 processors running at 2.66GHz.

### 6.1. Determinisation and completion

The columns in Table 1 show the overall effectiveness of three versions of the determinisation algorithm, for determinisation and determinisation with completion (+compl). DFTA is the textbook algorithm generating all transitions

---

<sup>2</sup>The code is available at <https://github.com/bishoksan/DFTA>

	DFTA		DFTA-opt		DFTA-opt-dc	
		+compl		+compl		+compl
solved	112	109	14672	14670	14672	14669
t/o	14584	14587	22	24	22	25
avg. secs.	119.9	119.9	0.30	0.37	0.30	0.34
% solved	0.76%	0.74%	99.85%	99.84%	99.85%	99.83%

Table 1: Average time (in seconds) for determinisation of 14,694 benchmark problems, with and without completion (timeout 120 seconds). DFTA = textbook algorithm; DFTA-opt = optimised algorithm without don’t care detection; DFTA-opt-dc = optimised algorithm with don’t care detection

explicitly, DFTA-opt is the optimised algorithm returning product form, without detection of don’t care arguments, and DFTA-opt-dc is with detection of don’t cares.

The first notable point is that the textbook algorithm is able to solve less than 1% of the problems while the optimised algorithms solve nearly all of them. The running time of the textbook algorithm is far slower even considering only those problems that it could solve.

The size of the input and output DFTAs is summarised in Table 2. The size of the set of states and transitions of the input FTA are  $|Q|$  and  $|\Delta|$ . The number of DFTA states is  $|Q_d|$  and the number of product transitions in the DFTA is  $|\Delta_\Pi|$ . For completed DFTAs, the precise size of the set of transitions  $\Delta_d$  depends only on the signature and can be calculated; this is shown in the table as  $|\Delta_d|$ . For non-complete DFTAs we cannot usually directly enumerate  $\Delta_d$  since it is too large. However, we estimate its size by summing the product of the sizes of the product states in each transition. (However, the same transition could be represented by more than one product transition so it is an over-estimate and we show it in brackets in Table 2). The comparison of the sizes and their distribution of  $\Delta_d$  and  $\Delta_\Pi$  is shown in the scatter-plots in Figure 6. Note that the scale on the vertical axes is logarithmic, and that while the majority of the benchmarks have  $\Delta_d$  with size over  $10^{12}$ , the sizes of  $\Delta_\Pi$  lie between 10 and  $10^4$ . This is a dramatic difference.

It can immediately be seen that the number of DFTA states is on average only slightly greater than the number of input FTA states, as discussed in Section 4. The average size of the set of transitions in completed DFTAs is extremely large, and shows immediately why the textbook algorithm fails. Regarding completion, the size of the set of states of both input and output automata is increased by one. The size of the input set of transitions is increased by the size of  $\Delta_{any}$  which is  $|\Sigma|$ .

Determinisation with completion usually results in a larger set of product transitions than determinisation alone. Without don’t cares, the average increase is a factor of about 9 (13,908 compared to 1,563). With don’t cares, the average increase factor is only about 2.5. However, in the optimised algorithms, completion hardly increases the running time, nor is detection of don’t cares a

	DFTA-opt		DFTA-opt-dc	
		+compl		+compl
average $ Q $	58.67	58.67	58.67	58.52
average $ \Delta $	293.58	307.66	293.58	306.23
average $ \Sigma $	15.5	15.5	15.5	15.5
average $ Q_d $	110.10	102.08	110.18	96.55
average $ \Delta_d $	$(2.23 \times 10^6)$	$2.93 \times 10^{18}$	$(2.25 \times 10^6)$	$2.93 \times 10^{18}$
average $ \Delta_\Pi $	1563.13	13908.79	1565.58	3817.90

Table 2: Size statistics for input and output of optimised DFTA algorithms (solved problems only).

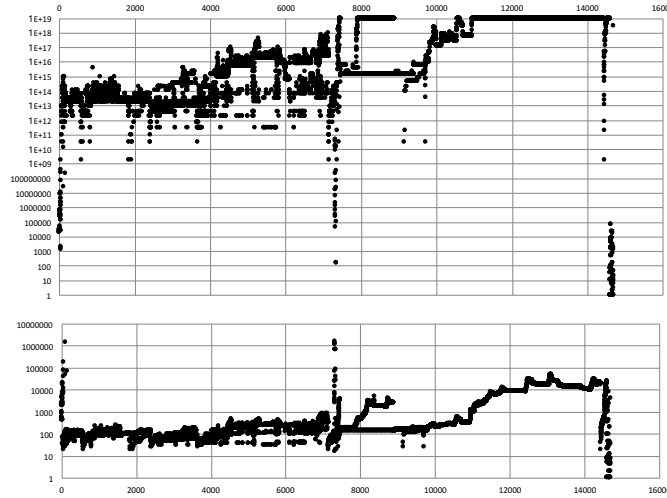


Figure 6: Distribution of size of  $\Delta_d$  and  $\Delta_\Pi$  for 14,672 complete determinised automata.

significant overhead.

## 6.2. Comparison with other FTA implementations

We have investigated the implementation of determinisation and completion (or closely related operations) in other available FTA libraries, namely Timbuk<sup>3</sup> [2], MONA<sup>4</sup> [4], the VATA tree automaton library<sup>5</sup> [5] and Autowrite<sup>6</sup> [6].

Timbuk provides a determinisation operation, which however is not optimised since determinisation plays no role in the main applications areas of Tim-

<sup>3</sup><http://people.irisa.fr/Thomas.Genet/timbuk/TimbukOnLine/>

<sup>4</sup><http://www.brics.dk/mona/>

<sup>5</sup><http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/>

<sup>6</sup><http://dept-info.labri.fr/~idurand/trag/>

buk, which are mainly the verification of cryptographic protocols<sup>7</sup>. We have not made an extensive comparison since the implementation seem to perform at best the same as our textbook implementation. Using the online version of Timbuk, the complementation of a medium-sized example with  $|Q| = 53$  and  $|\Delta| = 174$  runs out of resources, while our textbook implementation takes 7.5 seconds to produce the result with  $|\Delta_d| = 23,535$ . On the other hand, the optimised algorithm DFTA-opt computes the result in 50 milliseconds, yielding 501 product transitions.

MONA uses a technique called “guided tree automata” to optimise operations on tree automata. Guided tree automata are applied to FTAs that are already determinised by the user, and thus MONA does not explicitly offer a determinisation operation. Certain operations in MONA (for example projection) make the automata temporarily non-deterministic, and so these operations incorporate a determinisation operation to restore deterministic form. However, this operation handles only the restricted form of non-deterministic FTA that can arise, and is not directly accessible for testing, and so we are unable to make any direct comparison of our algorithm with MONA.

We made more extensive comparisons with VATA and Autowrite. The VATA library contains an FTA complementation operation. Complementation in VATA is not carried out by the classical procedure of determinisation and completion, followed by switching the accepting and non-accepting states of the output DFTA. However, it is the closest comparable operation. We took a set of 93 FTAs from the ARTMC system (Abstract Regular Tree Model Checking) benchmarks from the VATA benchmark library (which were included in the 14,672 previously mentioned). This is the set in which most of the timeouts of our optimised algorithm occur and therefore provide the most challenging comparison. Figure 7 shows the results of running VATA, DFTA and DFTA-opt (with completion) on these, with a timeout of 120 seconds. It can be seen that DFTA-opt solves about 75% of these, the majority of them fast, while VATA succeeds on only 5 within the time limit, comparable to our implementation of the textbook algorithm, which solves 15 within the time limit. In summary, from this sample it appears that VATA’s complementation performs at best as well as the textbook algorithm, but worse than the optimised algorithm for determinisation with completion.

The Autowrite tool is part of the TRAG system providing a library for analysis of term rewriting systems and graphs. Tree automata operations play an important role and thus the implementation of tree automata has been done with care [6]. The operations include determinisation and complementation. However, complementation does not generate all the transitions explicitly and the computation is essentially the same as determinisation. Therefore we compared with our optimised algorithm for determinisation without completion. On the same set of 93 problems used for comparison with VATA, Autowrite’s determinisation performance is comparable to our implementation of the textbook

---

<sup>7</sup>T. Genet. Personal communication

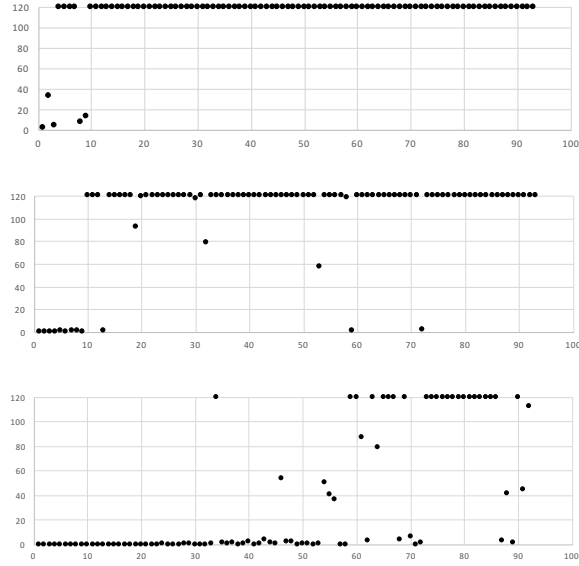


Figure 7: Comparison of time in seconds for complementation of 93 ARTMC benchmarks (timeout 120 secs.) with VATA (upper), DFTA-textbook (middle) and DFTA-opt (lower)

algorithm, solving 17 out of 93 within the 120 second time limit. The optimised algorithm for determinisation succeeded in 83 of the examples within the limit, with an average time (for the solved problems) of 17 seconds.

### 6.3. Performance on random FTAs

Figure 8 shows the result of running DFTA-opt on a set of 360 random FTAs.<sup>8</sup> These FTAs have 20 states, function symbols with arity 0, 2, 3 or 4, a transition density (the average number of different right-hand side states for a given left-hand side of a transition rule) ranging between 1.5 and 1.7 and an acceptance density (proportion of accepting states) ranging from 0.5 to 0.8. These were more challenging examples for all the algorithms tested. DFTA-opt solved 11.1% within the timeout as shown in Figure 8. The result for VATA’s complementation are not shown, but only 1.1% of the FTAs were complemented by VATA within the timeout.

The reason for the relatively poor performance of our algorithm (though still much better than the state of the art) on random FTAs is that the optimised algorithm’s success depends to some extent on function symbols having a natural “typing”. That is, a given state occurs in relatively few argument positions of a function symbol. This keeps the size of sets manipulated in the algorithm relatively small. In addition, as noted earlier, in many naturally occurring FTAs

<sup>8</sup>Obtained using a generator kindly supplied by R. Mayr and R. Almeida

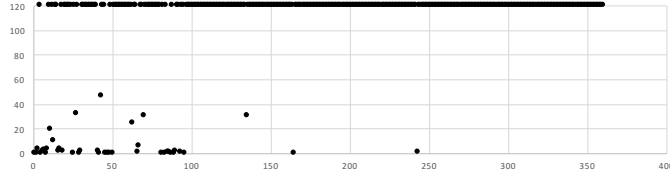


Figure 8: Time in seconds for determinisation and completion using DFTA-opt on 360 random FTAs (timeout 120 secs.).

the language associated with two states  $q_1$  and  $q_2$ , that is,  $L(q_1)$  and  $L(q_2)$ , tend to be either disjoint or one is included in the other. This keeps the number of determinised states down. Neither of these properties can be expected to hold in random FTAs.

## 7. Applications

We discuss two kinds of application: those where we need the set of transitions and those where we do not. For the first kind, applicability depends partly on whether the product form of transitions is directly usable. The product form is of little ultimate use if the transitions need to be explicitly enumerated in order to use them. Fortunately, it appears that the product form can often be efficiently processed directly.

### 7.1. FTA operations

Firstly, we note that other FTA operations can be adapted to handle product form directly, rather than expanding the products. For example, the intersection of FTAs, whose transitions are in product form, is easily performed. Let  $A^1, A^2$  be FTAs  $(Q_1, Q_1^f, \Sigma, \Delta_1)$  and  $(Q_2, Q_2^f, \Sigma, \Delta_2)$  respectively, where  $\Delta_1$  and  $\Delta_2$  are given in product form. Then the intersection  $A^1 \cap A^2$  is given by the FTA

$$(Q_1 \times Q_2, Q_1^f \times Q_2^f, \Sigma, \Delta_1 \times \Delta_2)$$

where the transition product  $\Delta_1 \times \Delta_2$  is the following set of product transitions.

$$\{f(R_1 \times S_1, \dots, R_n \times S_n) \rightarrow (q_1, q_2) \mid f(R_1, \dots, R_n) \rightarrow q_1 \in \Delta_1, f(S_1, \dots, S_n) \rightarrow q_2 \in \Delta_2\} .$$

The FTA minimisation procedure is defined on DFTAs and the operation is  $O(n^2)$  where  $n$  is the size of the input DFTA [7]. Given that our experiments show dramatic reduction of the size of DFTA when transitions are represented in product form, the adaptation of the minimisation algorithm to product form could greatly improve the scalability of the algorithm. We have performed preliminary experiments which confirm this expectation, but a detailed study of DFTA minimisation in product form is future work.

The difference of two FTAs can also be computed and represented in product form. The difference of two FTAs can be computed as follows [8]. Let  $A^1, A^2$  be FTAs  $(Q^1, Q_f^1, \Sigma, \Delta^1)$  and  $(Q^2, Q_f^2, \Sigma, \Delta^2)$  respectively, where we assume without loss of generality that  $Q^1$  and  $Q^2$  are disjoint. Let  $A^{1 \cup 2}$  be the union FTA  $(Q^1 \cup Q^2, Q_f^1 \cup Q_f^2, \Sigma, \Delta^1 \cup \Delta^2)$  and let  $(\mathcal{Q}', \mathcal{Q}_f', \Sigma, \Delta')$  be the determinisation of  $A^{1 \cup 2}$ . Let  $\mathcal{Q}^2 = \{Q' \in \mathcal{Q}' \mid Q' \cap Q_f^2 \neq \emptyset\}$ . Then  $A^1 \setminus A^2 = (\mathcal{Q}', \mathcal{Q}_f' \setminus \mathcal{Q}^2, \Sigma, \Delta')$ . Informally, the difference is constructed by computing the automaton accepting any term in the union  $L(A^1) \cup L(A^2)$  and then excluding any accepting state that can accept an element of  $L(A^2)$ .

### 7.2. Applications in program verification and analysis

Gallagher *et al.* [9, 10] showed that program properties of interest in analysis of logic program can be formulated as sets of terms defined by an FTA on the program signature and that a precise abstract domain for static analysis could then be constructed by determinising and completing the FTA. The algorithm presented in detail here was first developed in the context of that work. The approach was made practical by encoding the product transitions directly as binary decision diagrams (BDDs), via a translation to Datalog form [11] thus avoiding the need to enumerate transitions explicitly.

Recently the FTA difference construction mentioned above was used to implement a refinement procedure for Horn clause verification [8]. An FTA is built to represent the set of derivations in a given set of Horn clauses. Infeasible traces can be eliminated from the FTA by application of the automata difference algorithm, which is then used to construct a new set of Horn clauses in which the infeasible trace cannot arise.

Determinisation and completion are usually needed to form the complement of an FTA, though other algorithms exist, for example implemented in the VATA library to which we compared in Section 6. Applications in verification and analysis using tree automata to represent set of states could benefit from the availability of a practical complementation algorithm, e.g. [12, 13, 2, 14, 15]. Note that we obtain the complement with its transitions in product form, simply by switching the accepting and non-accepting states of the completed DFTA, and therefore further exploitation depends on avoiding the need to expand the product transitions.

### 7.3. Applications not using the set of transitions

For the second class of applications, that is, those for which transitions are not needed, we note that DFTA states encode useful information in themselves, making use of the disjointness of the sets  $\{L(Q_i) \mid Q_i \text{ is a DFTA state}\}$  and Lemma 2. For such applications, the optimised algorithm could be terminated without executing the transition generation step. We identify three applications referring only to the DFTA states.

1. Firstly, an FTA  $A$  over a signature  $\Sigma$  is *universal* if  $L(A) = \text{Term}(\Sigma)$ . This is true if in the determinised, completed DFTA, every state is an accepting

state, that is, it intersects with the set of accepting states of  $A$ . Note that a version of the algorithm checking universality could be terminated with a negative answer as soon as a counterexample state was generated.

2. Secondly, the *emptiness of intersections* of input FTA states can be checked using the DFTA states. Let  $q_1, \dots, q_n$  be FTA states. Then  $L(q_1) \cap \dots \cap L(q_n)$  is nonempty if and only if the corresponding DFTA includes a state containing  $q_1, \dots, q_n$ .
3. Finally we consider the problem of *FTA inclusion*. The classical approach to checking containment of FTA  $A_1$  in FTA  $A_2$  is to check the emptiness of  $A_1 \cap \bar{A}_2$  which involves the complementation of  $A_2$ . The procedure for constructing the difference FTA given above functions as an inclusion check. That is, to check whether  $L(A_1) \subseteq L(A_2)$  construct the states of the difference automaton  $A_2 \setminus A_1$ , as described above. Then  $L(A_1) \subseteq L(A_2)$  if and only if the set of accepting states in the DFTA of  $A_2 \setminus A_1$  is empty. As with universality checking, the algorithm can terminate with a negative answer as soon as a counterexample state is generated. Other containment algorithms not requiring complementation have been presented [16, 17] and an algorithm based on antichains [18]. The FTA inclusion-checking algorithms based on antichains [18] are in general more efficient, but our experiments show that the DFTA-based algorithm is comparable for cases where the answer is negative (in other words, a counterexample is found fast). However, in general the antichain algorithm is much more effective when inclusion holds.

## 8. Discussion and Related Work

The algorithm presented in this paper was sketched by Gallagher *et al.* in [10], including the concept of product transitions. Otherwise, we do not know of other attempts to design practical algorithms for determinisation. Previous work that used tree automata as a modelling formalism commented on the impracticality of handling complementation, due to the complexity of the determinisation and completion algorithm [13, 19]. Available libraries for tree automata manipulation seem to implement the textbook algorithm [2, 4, 5] for determinisation. It would be interesting to investigate whether the technique of “guided tree automata” in MONA could be modified to handle product transitions.

Efficient algorithms for FTA inclusion checking that avoid the need for determinisation have been developed [18], but the operation of determinisation is still required for many other purposes.

The key aspect of the optimised algorithm is the fact that the output is generated directly in a compact form (product transitions), which can be used directly in further processing. The space savings can be exponential, though the worst case remains the same. The problem of FTA determinisation is inherently intractable. As in other domains of formal reasoning, such as Boolean functions, great progress can nevertheless be achieved by finding compact representations



such as BDDs [20] that work well on a wide range of practical cases. Product form offers no guarantee of efficiency, but for a wide range of practical cases the compact representation makes the determinisation algorithm far more scalable.

## 9. Conclusion and Future Work

The contribution of this paper is that it is the first work to our knowledge that shows that determinisation and completion, previously considered almost hopeless cases for anything but very small FTAs, can be performed for a wide range of FTAs arising in practical applications.

There remains interesting work to do both on the algorithm itself and its applications. Firstly, there seem to be opportunities for optimisation of the critical inner loop of the algorithm generating the DFTA states. A state can be generated many times, and it seems likely that there are conditions on the elements of the  $\Phi$  and  $\Psi$  arrays in the algorithm that could be checked in order to avoid this. The challenge is to simplify the checks sufficiently to make them worthwhile as an optimisation. Perhaps a completely different representation of the  $\Phi$  and  $\Psi$  arrays, such as some Boolean encoding, is needed. We are actively investigating this.

Secondly, we are looking at other applications of the algorithm. The original motivating application, that of logic program analysis, is still interesting, since Horn clauses (pure logic programs) are increasingly used as a representation language for a variety of other languages and computational formalisms. Essentially the same analysis problems arise in term rewriting systems, where a system state is represented by a term, and an FTA expresses state properties of interest.

## Acknowledgements

We would like to thank Kim Steen Henriksen and Gourinath Banda for discussions in the early stages of this work.

## References

- [1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, Available on: <http://www.grappa.univ-lille3.fr/tata>, release October, 12th 2007 (2007).
- [2] T. Genet, V. V. T. Tong, Reachability analysis of term rewriting systems with Timbuk., in: R. Nieuwenhuis, A. Voronkov (Eds.), LPAR, Vol. 2250 of Lecture Notes in Computer Science, Springer, 2001, pp. 695–706.
- [3] A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar, Abstract regular tree model checking of complex dynamic data structures, in: K. Yi (Ed.), Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea,

- August 29-31, 2006, Proceedings, Vol. 4134 of Lecture Notes in Computer Science, Springer, 2006, pp. 52–70. doi:10.1007/11823230\_5.  
URL [http://dx.doi.org/10.1007/11823230\\_5](http://dx.doi.org/10.1007/11823230_5)
- [4] N. Klarlund, A. Møller, MONA Version 1.4 User Manual, BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus (January 2001).
  - [5] O. Lengál, J. Simáček, T. Vojnar, VATA: A library for efficient manipulation of non-deterministic tree automata, in: Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, 2012, pp. 79–94. doi:10.1007/978-3-642-28756-5\_7.  
URL [http://dx.doi.org/10.1007/978-3-642-28756-5\\_7](http://dx.doi.org/10.1007/978-3-642-28756-5_7)
  - [6] I. Durand, A tool for term rewrite systems and tree automata, Electr. Notes Theor. Comput. Sci. 124 (2) (2005) 29–49. doi:10.1016/j.entcs.2004.11.019.  
URL <https://doi.org/10.1016/j.entcs.2004.11.019>
  - [7] R. C. Carrasco, J. Daciuk, M. L. Forcada, An implementation of deterministic tree automata minimization, in: J. Holub, J. Zdárek (Eds.), Implementation and Application of Automata, 12th International Conference, CIAA 2007, Prague, Czech Republic, July 16-18, 2007, Revised Selected Papers, Vol. 4783 of Lecture Notes in Computer Science, Springer, 2007, pp. 122–129. doi:10.1007/978-3-540-76336-9\_13.  
URL [https://doi.org/10.1007/978-3-540-76336-9\\_13](https://doi.org/10.1007/978-3-540-76336-9_13)
  - [8] B. Kafle, J. P. Gallagher, Horn clause verification with convex polyhedral abstraction and tree automata-based refinement, Computer Languages, Systems & Structures 47 (2017) 2–18. doi:10.1016/j.cl.2015.11.001.  
URL <http://dx.doi.org/10.1016/j.cl.2015.11.001>
  - [9] J. P. Gallagher, K. S. Henriksen, Abstract domains based on regular types, in: V. Lifschitz, B. Demoen (Eds.), Proceedings of the International Conference on Logic Programming (ICLP’2004), Vol. 3132 of Springer-Verlag Lecture Notes in Computer Science, 2004, pp. 27–42.
  - [10] J. P. Gallagher, K. S. Henriksen, G. Banda, Techniques for scaling up analyses based on pre-interpretations, in: M. Gabbrielli, G. Gupta (Eds.), Proceedings of the 21st International Conference on Logic Programming, ICLP’2005, Vol. 3668 of Springer-Verlag Lecture Notes in Computer Science, 2005, pp. 280–296.
  - [11] J. Ullman, Principles of Knowledge and Database Systems; Volume 1, Computer Science Press, 1988.
  - [12] J. Elgaard, A. Møller, M. I. Schwartzbach, Compile-time debugging of C programs working on trees, in: Proc. Programming Languages and Systems, 9th European Symposium on Programming, ESOP ’00, Vol. 1782 of LNCS, Springer-Verlag, 2000, pp. 182–194.

- [13] D. Monniaux, Abstracting cryptographic protocols with tree automata, *Sci. Comput. Program.* 47(2-3) (2003) 177–202.
- [14] G. Feuillade, T. Genet, V. V. T. Tong, Reachability analysis over term rewriting systems, *J. Autom. Reasoning* 33 (3-4) (2004) 341–383.
- [15] E. Balland, Y. Boichut, P.-E. Moreau, T. Genet, Towards an efficient implementation of tree automata completion, in: *Algebraic Methodology and Software Technology, 12th International Conference, AMAST, 2008*, pp. 67–82.
- [16] T. Suda, H. Hosoya, Non-backtracking top-down algorithm for checking tree automata containment, in: *Implementation and Application of Automata, 10th International Conference, CIAA 2005, 2005*, pp. 294–306. doi:10.1007/11605157\\_25.  
URL [http://dx.doi.org/10.1007/11605157\\\_25](http://dx.doi.org/10.1007/11605157\_25)
- [17] H. Hosoya, J. Vouillon, B. C. Pierce, Regular expression types for XML, *ACM Trans. Program. Lang. Syst.* 27 (1) (2005) 46–90. doi:10.1145/1053468.1053470.  
URL <http://doi.acm.org/10.1145/1053468.1053470>
- [18] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, T. Vojnar, Antichain-based universality and inclusion testing over nondeterministic finite tree automata, in: O. H. Ibarra, B. Ravikumar (Eds.), *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings, Vol. 5148 of Lecture Notes in Computer Science, Springer, 2008*, pp. 57–67. doi:10.1007/978-3-540-70844-5\\_7.  
URL [http://dx.doi.org/10.1007/978-3-540-70844-5\\\_7](http://dx.doi.org/10.1007/978-3-540-70844-5\_7)
- [19] N. Heintze, Using bottom-up tree automaton to solve definite set constraints, unpublished. Presentation at Schloß Dagstuhl Seminar 9743 (1997).
- [20] R. E. Bryant, Graph-based algorithms for boolean function manipulation., *IEEE Trans. Computers* 35 (8) (1986) 677–691.